Ref #	Hits	Search Query	DBs	Default Operator	Plurals	Time Stamp
S77	4	huang.in. and nen-fu.in.	US-PGPUB; USPAT; EPO; JPO; IBM_TDB	OR	ON	2004/12/27 12:41
S78	156	longest near2 prefix and next adj hop and pointer	US-PGPUB; USPAT; EPO; JPO; IBM_TDB	OR	ON	2004/12/27 12:42
S79	95	S78 and offset	US-PGPUB; USPAT; EPO; JPO; IBM_TDB	OR	ON	2004/12/27 12:42
S80	19	S79 and (variable or varying or dynamic) near4 offset	US-PGPUB; USPAT; EPO; JPO; IBM_TDB	OR	ON	2004/12/27 12:42
S81	0	S80 and (@ad<"19990201" or @prad<"19990201")	US-PGPUB; USPAT; EPO; JPO; IBM_TDB	OR	ON	2004/12/27 12:43
S82	5	S79 and (@ad<"19990201" or @prad<"19990201")	US-PGPUB; USPAT; EPO; JPO; IBM_TDB	OR	ON	2004/12/27 12:43
S83	13	S78 and (@ad<"19990201" or @prad<"19990201")	US-PGPUB; USPAT; EPO; JPO; IBM_TDB	OR	ON	2004/12/27 12:51
S84	2	"6539369"	US-PGPUB; USPAT; EPO; JPO; IBM_TDB	OR	ON	2004/12/27 12:51
S85	26	"6192051"	US-PGPUB; USPAT; EPO; JPO; IBM_TDB	OR	ON	2004/12/27 12:51
S86	8	("5412654" | "5842224" | "5870739" | "5946679" | "6011795" | "6014659" | "6052683" | "6061712").PN.	US-PGPUB; USPAT; USOCR	OR	ON	2004/12/27 12:52

IEEE Xplore™ ◂ Back to Previous Page

IP lookups using multiway and multicolumn search

Lampson, B. Srinivasan, V. Varghese, G.
Microsoft, USA

This paper appears in: **INFOCOM '98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE**

Meeting Date: 03/29/1998 - 04/02/1998
Publication Date: 29 March-2 April 1998
Location: San Francisco, CA USA
On page(s): 1248 - 1256 vol.3
Volume: 3
Reference Cited: 18
Number of Pages: 3 vol. xxviii+1477
Inspec Accession Number: 6004784

Abstract:
IP address lookup is becoming critical because of increasing routing table size, speed, and traffic in the Internet. Our paper shows how binary search can be adapted for best **matching prefix** using two entries per prefix and by doing precomputation. Next we show how to improve the performance of any best **matching prefix** scheme using an initial array indexed by the first X bits of the address. We then describe how to take advantage of cache line size to do a multiway search with 6-way branching. Finally, we show how to extend the binary search solution and the multiway search solution for IPv6. For a database of N prefixes with address length W, naive binary search scheme would take O(W*logN); we show how to reduce this to O(W+logN) using multiple column binary search. Measurements using a practical (Mae-East) database of 30000 entries yield a worst case lookup time of 490 nanoseconds, five times faster than the Patricia trie scheme used in BSD UNIX. Our scheme is attractive for IPv6 because of small storage requirement (2N nodes) and speed (estimated worst case of 7 cache line reads)

Index Terms:
Internet search problems table lookup telecommunication network routing telecommunication traffic transport protocols 6-way branching BSD UNIX IP address lookup IP packet forwarding rates IPv6 Internet address length best **matching prefix** cache line size database encoding measurements multiple column binary search multiway search multiway search solution performance precomputation routing table size small storage requirement speed traffic

Documents that cite this document
Select link to view other documents in the database that cite this one.

IP Lookups using Multiway and Multicolumn Search

Butler Lampson, V Srinivasan and George Varghese
blampson@microsoft.com,cheenu@dworkin.wustl.edu,varghese@dworkin.wustl.edu

Abstract— *IP address lookup* **is becoming critical because of increasing routing table size, speed, and traffic in the Internet. Our paper shows how binary search can be adapted for best matching prefix using two entries per prefix and by doing precomputation. Next we show how to improve the performance of any best matching prefix scheme using an initial array indexed by the first X bits of the address. We then describe how to take advantage of cache line size to do a multiway search with 6-way branching. Finally, we show how to extend the binary search solution and the multiway search solution for IPv6. For a database of N prefixes with address length W, naive binary search scheme would take $O(W*logN)$; we show how to reduce this to $O(W+logN)$ using multiple column binary search. Measurements using a practical (Mae-East) database of 30000 entries yield a worst case lookup time of 490 nanoseconds, five times faster than the Patricia trie scheme used in BSD UNIX. Our scheme is attractive for IPv6 because of small storage requirement (2N nodes) and speed (estimated worst case of 7 cache line reads)**

Keywords— **Longest Prefix Match, IP Lookup**

I. Introduction

Statistics show that the number of hosts on the internet is tripling approximately every two years [oT]. Traffic on the Internet is also increasing exponentially. Traffic increase can be traced not only to increased hosts, but also to new applications (e.g., the Web, video conferencing, remote imaging) which have higher bandwidth needs than traditional applications. One can only expect further increases in users, hosts, domains, and traffic. The possibility of a global Internet with multiple addresses per user (e.g., for appliances) has necessitated a transition from the older Internet routing protocol (IPv4 with 32 bit addresses) to the proposed next generation protocol (IPv6 with 128 bit addresses).

High speed packet forwarding is compounded by increasing routing database sizes (due to increased number of hosts) and the increased size of each address in the database (due to the transition to IPv6). Our paper deals with the problem of increasing IP packet forwarding rates in routers. In particular, we deal with a component of high speed forwarding, *address lookup*, that is considered to be a major bottleneck.

When an Internet router gets a packet P from an input link interface, it uses the destination address in packet P to lookup a routing database. The result of the lookup provides an output link interface, to which packet P is forwarded. There is some additional bookkeeping such as updating packet headers, but the major tasks in packet forwarding are address lookup and switching packets between link interfaces.

George Varghese was supported by an ONR Young Investigator Award and NSF Research Award NCR-9628145

For Gigabit routing, many solutions exist which do fast switching within the router box [NMH97]. Despite this, the problem of doing lookups at Gigabit speeds remains. For example, Ascend's product [Asc] has *hardware* assistance for lookups and can take up to 3 μs for a single lookup in the worst case and 1 μs on average. However, to support say 5 Gbps with an average packet size of 512 bytes, lookups need to be performed in 800 nsec per packet. By contrast, our scheme can be implemented in *software* on an ordinary PC in a worst case time of 490 nsec.

The Best Matching Prefix Problem: Address lookup can be done at high speeds if we are looking for an *exact match* of the packet destination address to a corresponding address in the routing database. Exact matching can be done using standard techniques such as hashing or binary search. Unfortunately, most routing protocols (including OSI and IP) use hierarchical addressing to avoid scaling problems. Rather than have each router store a database entry for all possible destination IP addresses, the router stores address *prefixes* that represent a group of addresses reachable through the same interface. The use of prefixes allows scaling to worldwide networks.

The use of prefixes introduces a new dimension to the lookup problem: multiple prefixes may match a given address. If a packet matches multiple prefixes, it is intuitive that the packet should be forwarded corresponding to the *most specific* prefix or *longest* prefix match. IPv4 prefixes are arbitrary bit strings up to 32 bits in length as shown in Table I. To see the difference between the exact matching and best matching prefix, consider a 32 bit address A whose first 8 bits are 10001111. If we searched for A in the above table, exact match would not give us a match. However prefix matches are 100* and 1000*, of which the best matching prefix is 1000*, whose next hop is $L5$.

Prefix	Next hop
*	L9
001*	L1
0001*	L2
011111*	L3
100*	L4
1000*	L5
10001*	L6

TABLE I
A sample routing table

Paper Organization:

The rest of this paper is organized as follows. Section II describes related work and briefly describes our contribution. Section III contains our basic binary search scheme. Section IV describes a basic idea of using an array as a front end to reduce the number of keys required for binary search. Section V describes how we exploit the locality inherent in cache lines to do multiway binary search; we also describe measurements for a sample large IPv4 database. Section VII describes how to do multicolumn and multiway binary search for IPv6. We also describe some measurements and projected performance estimates. Section VIII states our conclusions.

II. Previous Work and Our Contributions

[MTW95] uses content-addressable memories(CAMs) for implementing best matching prefix. Their scheme uses a separate CAM for each possible prefix length. For IPv4 this can require 32 CAMs and 128 CAMs for IPv6, which is expensive.

The current NetBSD implementation [SW95], [Skl] uses a *Patricia Trie* which processes an address one bit at a time. On a 200 MHz pentium, with about 33,000 entries in the routing table, this takes 1.5 to 2.5 μ s on the average. These numbers will worsen with larger databases. [Skl] mentions that the expected number of bit tests for the patricia tree is 1.44 log N, where N is the number of entries in the table. For N=32000, this is over 21 bit tests. With memory accesses being very slow for modern CPUs, 21 memory accesses is excessive. Patricia tries also use skip counts to compress one way branches, which necessitates backtracking. Such backtracking slows down the algorithm and makes pipelining difficult.

Many authors have proposed tries of high radix [PZ92] but only for exact matching of addresses. OSI address lookups are done naturally using trie search 4 bits at a time [Per92] but that is because OSI prefix lengths are always multiples of 4. Our methods can be used to lookup OSI address lookups 8 bits at a time.

[NMH97] claims that it is possible to do a lookup in 200 nsec using SRAMs (with 10 nsec cycle times) to store the entire routing database. We note that large SRAMs are extremely expensive and are typically limited to caches in ordinary processors.

Caching is a standard solution for improving average performance. However, experimental studies have shown poor cache hit ratios for backbone routers[NMH97]. This is partly due to the fact that caches typically store whole addresses. Finally, schemes like Tag and Flow Switching suggest protocol changes to avoid the lookup problem altogether. These proposals depend on widespread acceptance, and do not completely eliminate the need for lookups at network boundaries.

In the last year, two new techniques [BCDP97], [WVTP97] for doing best matching prefix have been announced. The approach in [BCDP97] is based on compressing trie nodes so that they will fit into the cache. The approach in [WVTP97] is based on doing binary search on the ***possible prefix lengths.***

Another approach invented and patented by us based on prefix expansion [SV98] seems to be the simplest and fastest of the schemes we know for IPv4. [1] Detailed comparisons with other schemes are presented in [SV98].

Our Contributions:

In this paper, we start by showing how to modify binary search to do best matching prefix. Modified binary search requires two ideas: first, we treat each prefix as a range and encode it using the start and end of range; second, we arrange range entries in a binary search table and precompute a mapping between consecutive regions in the binary search table and the corresponding prefix.

Our approach is completely different from either [WVTP97] as we do binary search on the *number of possible prefixes* as opposed to *the number of possible prefix lengths.*. For example, the naive complexity of our scheme is $\log_2 N + 1$ memory accesses, where N is the number of prefixes; by contrast, the complexity of the [WVTP97] scheme is $\log_2 W$ hash computations plus memory accesses, where W is the length of the address in bits.

At a first glance, it would appear that the scheme in [WVTP97] would be faster (except potentially for hash computation, which is not required in our scheme) than our scheme, especially for large prefix databases. However, we show that we can exploit the locality inherent in processor caches and fast cache line reads using SDRAM or RDRAM to do multiway search in $\log_{k+1} N + 1$ steps, where $k > 1$. We have found good results using $k = 5$. By contrast, it appears to be impossible to modify the scheme in [WVTP97] to do multiway search on prefix lengths because each search in a hash table only gives two possible outcomes.

Further, for long addresses (e.g., 128 bit IPv6 addresses), the true complexity of the scheme in [WVTP97] is closer to $O(W/M) \log_2 W$, where M is the word size of the machine.[2] This is because computing a hash on a W bit address takes $O(W/M)$ time. By contrast, we introduce a multicolumn binary search scheme for IPv6 and OSI addresses that takes $\log_2 N + W/M + 1$. Notice that the W/M factor is additive and not multiplicative. Using a machine word size of $M = 32$ and an address width W of 128, this is a potential multiplicative factor of 4 that is avoided in our scheme.

The approach in [BCDP97] is based on compressing k-bit trie nodes and takes O(W/k) time. It differs entirely from our approach. While the approach in [BCDP97] has search times comparable to ours for IPv4 (around 500 nsec), our approach should scale better for IPv6 when W becomes 128.

We also describe a simple scheme of using an initial array as a front end to reduce the number of keys required to be searched in binary search. Essentially, we partition the original database according to every possible combination of the first X bits. Our measurements use $X = 16$. Since

[1] Details of this method are not in the public domain yet due to the patenting process.

[2] The scheme in [WVTP97] starts by doing a hash of $W/2$ bits; it can then do a hash on $3W/4$ bits, followed by $7W/8$ bits etc. Thus in the worst case, each hash may operate on roughly $3W/4$ bits.

the number of possible prefixes that begin with a particular combination of the first X bits is much smaller than the total number of prefixes, this is a big factor in practice.

Our paper describes the results of several other measurements of speed and memory usage for our implementations of these two schemes. The measurements allow us to isolate the effects of individual optimizations and architectural features of the CPUs we used. We describe results using a publically available routing database (Mae-East NAP) for IPv4, and by using randomly chosen 128 bit addresses for IPv6.

Our measurements show good results. Measurements using the (Mae-East) database of 30000 entries yield a worst case lookup time of 490 nanoseconds, five times faster than the performance of the Patricia trie scheme used in BSD UNIX used on the same database. We also estimate the performance of our scheme for IPv6 using a special SDRAM or RDRAM memory (which is now commercially available though we could not obtain one in time to do actual experiments). This memory allows fast access to data within a page of memory, which enables us to speed up multiway search. Thus we estimate a worst case figure of 7 cache line reads for a large database of IPv6 entries.

Please note that in the paper, by *memory reference* we mean accesses to the main memory. Cache hits are not counted as memory references. So, if a cache line of 32 bytes is read, then accessing two different bytes in the 32 byte line is counted as one memory reference. This is justifiable, as a main memory read has an access time of 60 nsec while the on-chip L1 cache can be read at the clock speed of 5 nsec on an Intel Pentium Pro. With SDRAM or RDRAM, a cache line fill is counted as one memory access. With SDRAM a cache line fill is a burst read with burst length 4. While the first read has an access time of 60 nsec, the remaining 3 reads have access times of only 10 nsec each [Mic]. With RDRAM, an entire 32 byte cache line can be filled in 101 nsec [Ram].

III. Adapting Binary search for Best Matching Prefix

Binary search can be used to solve the best matching prefix problem, but only after several subtle modifications. Assume for simplicity in the examples, that we have 6 bit addresses and three prefixes 1*, 101*, and 10101*. First, binary search does not work with variable length strings. Thus the simplest approach is to pad each prefix to be a 6 bit string by adding zeroes. This is shown in Figure 1.

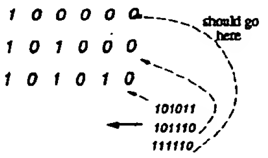


Fig. 1. Placing the three prefixes 1*, 101*, and 10101* in a binary search table by padding each prefix with 0's to make 6 bit strings and sorting the resulting strings. Note that the addresses 101011, 101110, and 111110 all end up in the same region in the binary search table

Now consider a search for the three 6 bit addresses 101011, 101110, and 111110. Since none of these addresses are in the table, binary search will fail. Unfortunately, on a failure all three of these addresses will end up at the end of the table because all of them are greater than 101010, which is the last element in the binary search table. Notice however that each of these three addresses (see Figure 1) has a different best matching prefix.

Thus we have two problems with naive binary search: first, when we search for an address we end up far away from the matching prefix (potentially requiring a linear search); second, multiple addresses that match to different prefixes, end up in the same region in the binary table (Figure 1).

Encoding Prefixes as Ranges:

To solve the second problem, we recognize that a prefix like 1* is really a range of addresses from 100000 to 111111. Thus instead of encoding 1* by just 100000 (the start of the range), we encode it using both the start and end of range. Thus each prefix is encoded by two full length bit strings. These bit strings are then sorted. The result for the same three prefixes is shown in Figure 2. We connect the start and end of a range (corresponding to a prefix) by a line in Figure 2. Notice how the ranges are nested. If we now try to search for the same set of addresses, they each end in a different region in the table. To be more precise, the search for address 101011 ends in an exact match. The search for address 101110 ends in a failure in the region between 101011 and 101111 (Figure 2), and the search for address 111110 ends in a failure in the region between 101111 and 111111. Thus it appears that the second problem (multiple addresses that match different prefixes ending in the same region of the table) has disappeared. Compare Figure 1 and Figure 2.

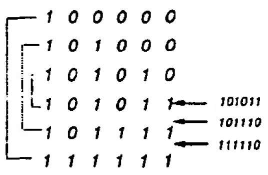


Fig. 2. We now encode each prefix in the table as a range using two values: the start and end of range. This time the addresses that match different prefixes end up in different ranges.

To see that this is a general phenomenon, consider Figure 3. The figure shows an arbitrary binary search table after every prefix has been encoded by the low (marked **L** in Figure 3) and its high points (marked **H**) of the corresponding range. Consider an arbitrary position indicated by the solid arrow. If binary search for address A ends up at this point, which prefix should we map A to? It is easy to see the answer visually from Figure 3. If we start from the point shown by the solid arrow and we go back up the table, the prefix corresponding to A is the first **L** that is not followed by a corresponding **H** (see dotted arrow in Figure 3.)

Why does this work? Since we did not encounter an **H** corresponding to this **L**, it clearly means that A is contained in the range corresponding to this prefix. Since this is the first such **L**, this is the smallest such range. Essentially, this works because the best matching prefix has been translated to the problem of finding the *narrowest enclosing range.*

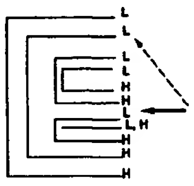


Fig. 3. Why each range in the modified binary search table maps to a unique prefix.

A. Using Precomputation to Avoid Search

Unfortunately, the solution depicted in Figure 2 and Figure 3 does not solve the first problem: notice that binary search ends in a position that is far away (potentially) from the actual prefix. If we were to search for the prefix (as described earlier), we could have a linear time search.

However, the modified binary search table shown in Figure 3 has a nice property we can exploit. *Any region in the binary search between two consecutive numbers corresponds to a unique prefix.* As described earlier, the prefix corresponds to the first **L** before this region that is not matched by a corresponding **H** that also occurs before this region. Similarly, every exact match corresponds to a unique prefix.

But if this is the case, we can precompute the prefix corresponding to each region and to each exact match. This can potentially slow down insertion. However, the insertion or deletion of a new prefix should be a rare event (the next hop to reach a prefix may change rapidly, but the addition of a new prefix should be rare) compared to packet forwarding times. Thus slowing down insertion costs for the sake of faster forwarding is a good idea. Essentially, the idea is to add the dotted line pointer shown in Figure 3 to every region.

The final table corresponding to Figure 3 is shown in Figure 5. Notice that with each table entry E, there are two precomputed prefix values. If binary search for address A ends in a failure at E, it is because $A > E$. In that case, we use the $>$ pointer corresponding to E. On the other hand, if binary search for address A ends in a match at E, we use the $=$ pointer.

Notice that for an entry like 101011, the two entries are different. If address A ends up at this point and is greater than 101011, clearly the right prefix is $P2$ = 101*. On the other hand, if address A ends up at this point with equality, the correct prefix is $P3$ = 10101*. Intuitively, if an address A ends up equal to the high point of a range R, then A fall within the range R; if A ends up greater than the high point of range R, then A falls within the smallest range that encloses range R.

Our scheme is somewhat different from the description in [Per92]. We use two pointers per entry instead of just one pointer. The description of our scheme in [Per92] suggests padding every address by an extra bit; this avoids the need for an extra pointer but it makes the implementation grossly inefficient because it works on 33 bit (i.e., for IPv4) or 129 bit (i.e., for IPv6) quantities. If there are less than 2^{16} different choices of next hop, then the two pointers can be packed into a 32 bit quantity, which is probably the minimum storage needed.

							>	=
P1)	1	0	0	0	0	0	P1	P1
P2)	1	0	1	0	0	0	P2	P2
P3)	1	0	1	0	1	0	P3	P3
	1	0	1	0	1	1	P2	P3
	1	0	1	1	1	1	P1	P2
	1	1	1	1	1	1	-	P1

Fig. 4. The final modified binary search table with precomputed prefixes for every region of the binary table. We need to distinguish between a search that ends in a success at a given point (= pointer) and search that ends in a failure at a given point (> pointer).

B. Insertion into a Modified Binary Search Table

The simplest way to build a modified binary search table from scratch is to first sort all the entries, after marking each entry as a high or a low point of a range. Next, we process the entries, using a stack, from the lowest down to the highest to precompute the corresponding best matching prefixes. Whenever we encounter a low point (L in the figures), we stack the corresponding prefix; whenever we see the corresponding high point, we unstack the prefix. Intuitively, as we move down the table, we are keeping track of the currently active ranges; the top of the stack keeps track of the innermost active range. The prefix on top of the stack can be used to set the $>$ pointers for each entry, and the $=$ pointers can be computed trivially. This is an $O(N)$ algorithm if there are N prefixes in the table.

One might hope for a faster insertion algorithm if we had to only add (or delete) a prefix. First, we could represent the binary search table as a binary tree in the usual way. This avoids the need to shift entries to make room for a new entry. Unfortunately, the addition of a new prefix can affect the precomputed information in $O(N)$ prefixes. This is illustrated in Figure 5. The figure shows an outermost range corresponding to prefix P; inside this range are $N-1$ smaller ranges (prefixes) that do not intersect. In the regions not covered by these smaller prefixes, we map to P. Unfortunately, if we now add Q (Figure 5), we cause all these regions to map to Q, an $O(N)$ update process.

Thus there does not appear to be any update technique that is faster than just building a table from scratch. Of course, many insertions can be batched; if the update process falls behind, the batching will lead to more efficient updates.

IV. Precomputed 16 bit prefix table

We can improve the worst case number of memory accesses of the basic binary search scheme with a precomputed table of best matching prefixes for the first Y bits.

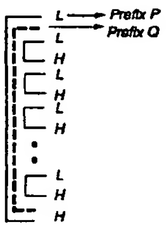


Fig. 5. Adding a new prefix Q (dotted line) can cause all regions between an **H** and an **L** to move from Prefix P to Prefix Q.

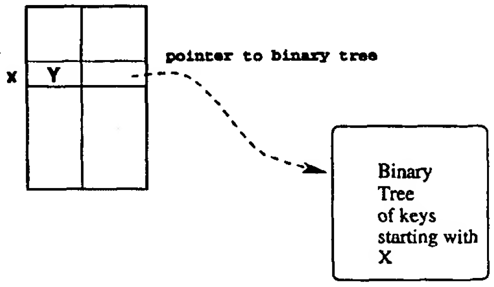


Fig. 6. The array element with index X will have the best matching prefix of X (say Y) and a pointer to a binary tree/table of all prefixes that have X as a prefix.

The main idea is to effectively partition the single binary search table into multiple binary search tables for each value of the first Y bits. This is illustrated in Figure 6. We choose $Y = 16$ for what follows as the table size is about as large as we can afford, while providing maximum partitioning.

Without the initial table, the worst case possible number of memory accesses is $log_2 N + 1$, which for large databases could be 16 or more memory accesses. For a sample database, this simple trick of using an array as a front end reduces the maximum number of prefixes in each partitioned table to 336 from the maximum value of over 30,000.

The best matching prefixes for the first 16 bit prefixes can be precomputed and stored in a table. This table would then have $Max = 65536$ elements. For each index X of the array, the corresponding array element stores best matching prefix of X. Additionally, if there are prefixes of longer length with that prefix X, the array element stores a pointer to a binary search table/tree that contains all such prefixes. Insertion, deletion, and search in the individual binary search tables is identical to the technique described earlier in Section III.

Figure 7 shows the distribution of the number of keys that would occur in the individual binary search trees for a publically available IP backbone router database [Mer] after going through an initial 16 bit array. The largest number of keys in any binary table is found to be 336, which leads to a worst case of 10 memory accesses.

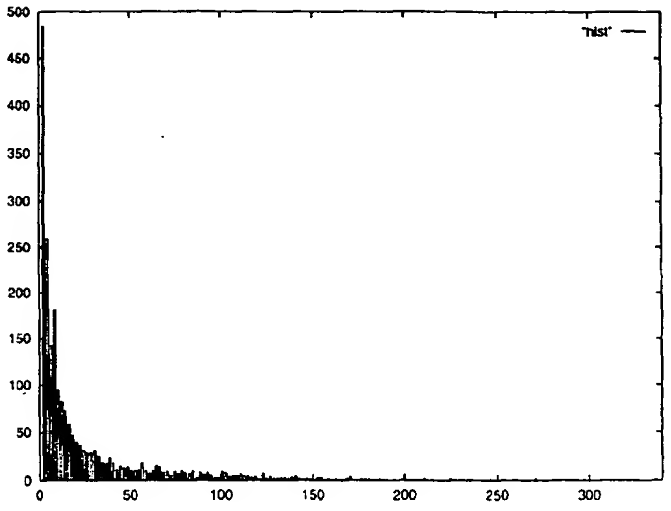


Fig. 7. For each 16 bit prefix X, let N(X) be the number of prefixes that have X as a prefix. The histogram shows the distribution of N(X) for the Mae-East NAP Routing database [Mer]. The horizontal axis represents N(X) and the vertical axis represents the number of tables with a given value of N(X). Thus the peak of the histogram says that there are 484 binary search tables with only 2 keys. There is only 1 binary search table with the worst case number of 336 keys.

V. Multiway binary search: Exploiting the cache line

Todays processors have wide cache lines. The Intel Pentium Pro has a cache line size of 32 bytes. Main memory is usually arranged in a matrix form, with rows and columns. Accessing data given a random row address and column address has an access time of 50 to 60 nsec. However, using SDRAM or RDRAM, filling a cache line of 32 bytes is much faster, which is a burst access to 4 contiguous 64 bit DRAM locations, is much faster than accessing 4 random DRAM locations. When accessing a burst of contiguous columns in the same row, while the first piece of data would be available only after 60 nsec, further columns would be available much faster. SDRAMs (Synchronous DRAMs) are available (at \$205 for 8MB [Sim]) that have a column access time of 10 nsec. Timing diagrams of micron SDRAMs are available through [Mic]. RDRAMs [Ram] are available that can fill a cache line in 101 nsec. The Intel Pentium pro has a 64 bit data bus and a 256 bit cacheline [Inta]. Detailed descriptions of main memory organization can be found in [HP96].

The significance of this observation is that it pays to restructure data structures to improve locality of access. To make use of the cache line fill and the burst mode, keys and pointers in search tables can be laid out to allow multiway search instead of binary search. This effectively allows us to reduce the search time of binary search from $\log_2 N$ to $\log_{k+1} N$, where k is the number of keys in a search node. The main idea is to make k as large as possible so that a single search node (containing k keys and $2k + 1$ pointers) fits into a single cache line. If this can be arranged, an access to the first word in the search node will result in the entire node being prefetched into cache. Thus the accesses to the remaining keys in the search node are much cheaper than a memory access.

We did our experiments using a Pentium Pro; the parameters of the Pentium Pro resulted in us choosing $k = 5$ (i.e, doing a six way search). For our case, if we use k keys per node, then we need $2k + 1$ pointers, each of which is a 16 bit quantity. So in 32 bytes we can place 5 keys and hence can do a 6-way search. The initial full array of 16 bits followed by the 6-way search is depicted in Figure 8.

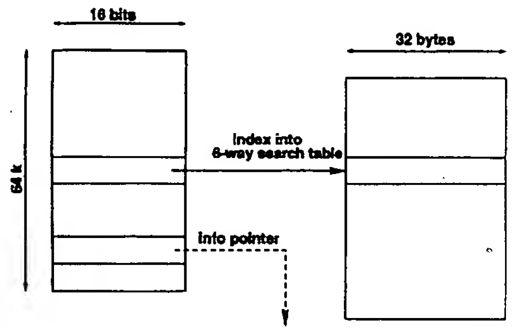


Fig. 8. The initial 16 bit array, with pointers to the corresponding 6-way search nodes.

This shows that the worst case (for the Mae East database after using a 16 bit initial array) has 336 entries leading to a worst case of 4 memory accesses (since 6^4 =1296 takes only 4 memory accesses when doing a 6-way search).

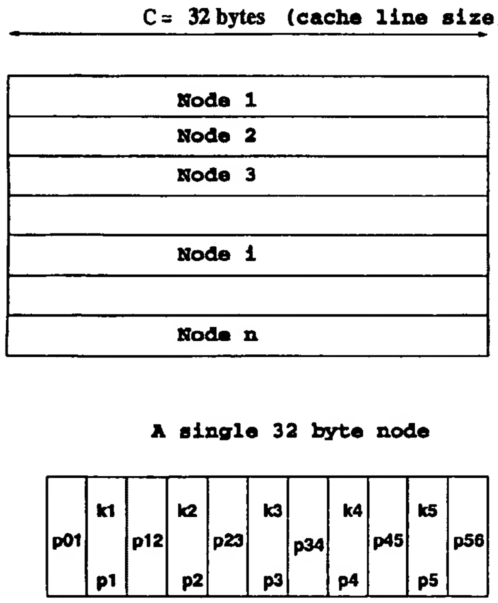


Fig. 9. The structure of the 6-way search node. There are k keys and $2k + 1$ pointers.

Each node in the 6-way search table has 5 keys k_1 to k_5, each of which is 16 bits. There are *equal to* pointers p_1 to p_5 corresponding to each of these keys. Pointers p_{01} to p_{56} correspond to ranges demarcated by the keys. This is shown in Figure 9 . Among the keys we have the relation $k_i \leq k_{i+1}$. Each pointer has a bit which says it is an *information* pointer or a next node pointer.

A. Search

The following search procedure can be used for both IPv4 and IPv6. For IPv6, 32 bit keys can be used instead of 16 bits.

1. Index into the first 16 bit array using the first 16 bits of the address.
2. If the pointer at the location is an *information* pointer, return it. Otherwise enter the 6-way search with the initial node given by the pointer, and the key being the next 16 bits of the address.
3. In the current 6-way node locate the position of the key among the keys in the 6-way node. We use binary search among the keys within a node. If the key equals any of the keys key_i in the node, use the corresponding pointer ptr_i. If the key falls in any range formed by the keys, use the pointer $ptr_{i,i+1}$. If this pointer is an *information* pointer, return it; otherwise repeat this step with the new 6-way node given by the pointer.
In addition, we allow multicolumn search for IPv6 (see Section VII) as follows. If we encounter an *equal to* pointer, the search shifts to the next 16 bits of the input address. This feature can be ignored for now and will be understood after reading Section VII.

As the data structure itself is designed with a node size equal to a cache line size, good caching behavior is a consequence. All the frequently accessed nodes will stay in the cache. To reduce the worst case access time, the first few levels in a worst case depth tree can be cached.

VI. Measurements and Comparison for IPv4

We used a *Pentium Pro* [Intb] based machine, with a 200 MHz clock (cost under 5000 dollars). It has a 8 KByte four-way set-associative primary instruction cache and a 8 KByte dual ported two-way set associative primary data cache. The L2 cache is 256 KBytes of SRAM that is coupled to the core processor through a full clock-speed, 64-bit, cache bus.

We used a practical routing Table with over 32000 entries that we obtained from [Mer] for our experiments. Our tables list results for the BSD Radix Trie implementation (extracted from the BSD kernel into user space), binary search (Bsearch) and 6-way search.

Repeated lookup of a single address: After adding the routes in the route database VI, random IP addresses were generated and a lookup performed 10 million times for each such address. We picked 7 of these results to display in Table II.

Average search time: 10 million IP addresses were generated and looked up, assuming that all IP addresses were equally probable. It was found that the average lookup time was 130 nanoseconds.

Memory Requirements and Worst case time

The memory requirement for the 6-way search is less than that for basic binary search! Though at first this looks counter-intuitive, this is again due to the initial 16

Patricia	Basic binary search	16 bit +binary search	16 bit +6 way search
Time (nsec)	Time (nsec)	Time (nsec)	Time (nsec)
1530	1175	730	490
1525	990	620	490
1450	1140	470	390
2585	1210	400	300
1980	1440	330	210
810	1220	90	95
1170	1310	90	90

TABLE II

Time taken for single address lookup on a Pentium pro. Several addresss were searched and the search times noted. Shown in the table are addresses picked to illustrate the variation in time of the 16 bit initial table+6-way search method. Thus the first two rows correspond to the maximum depth of the search tree while the last two rows correspond to the minimum depth (i.e, no prefixes in search table).

	Patricia	Basic Binary Search	16 bit table +binary	16 bit table +6 way
Mem for building(MB)	3.2	3.6	1	1
Mem for searchable structure (MB)	3.2	1	0.5	0.7
Worst case search(nsec)	2585	1310	730	490
Worst case faster than Patricia by	1	2	3.5	5

TABLE III

Memory Requirement and Worst case time

bit array. While the keys used in the regular binary search are 32 bits and the pointers involved are also 32 bits, in the 16 bit table followed by the 6-way search case, both the keys and the pointers are 16 bits.

From Table III we can see that the initial array improves the performance of the binary search from a worst case of 1310 nsec to 730 nsec; multiway search further improves the search time to 490 nsec.

Instruction count:

The static instruction count for the search using a full 16 bit initial table followed by a 6-way search table is less than 100 instructions on the Pentium Pro. We also note that the gcc compiler uses only 386 instructions and does not use special instructions available in the pentium pro, using which it might be possible to further reduce the number of instructions.

VII. Using Multiway and Multicolumn Search for IPv6

In this section we describe the problems of searching for identifiers of large width (e.g., 128 bit IPv6 address or 20 byte OSI addresses). We first describe the basic ideas behind multicolumn search and then proceed to describe an implementation for IPv6 that uses both multicolumn and multiway search. We then describe sample measurements using randomly generated IPv6 addresses.

A. Multicolumn Binary Search of Large Identifiers

The scheme we have just described can be implemented efficiently for searching 32 bit IPv4 addresses. Unfortunately, a naive implementation for IPv6 can lead to inefficiency. Assume that the word size M of the machine implementing this algorithm is 32 bits. Since IPv6 addresses are 128 bits (4 machine words), a naive implementation would take $4 \cdot \log_2(2N)$ memory accesses. For a reasonable sized table of around 32,000 entries this is around 60 memory accesses!

In general, suppose each identifier in the table is W/M words long (for IPv6 addresses on a 32 bit machine, $W/M = 4$). Naive binary search will take $W/M \cdot \log N$ comparisons which is expensive. Yet, this seems obviously wasteful. If all the identifiers have the same first $W/M - 1$ words, then clearly $\log N$ comparisons are sufficient. We show how to modify Binary Search to take $\log N + W/M$ comparisons. It is important to note that this optimization we describe can be *useful for any use of binary search on long identifiers*, not just the best matching prefix problem.

The strategy is to work in columns, starting with the most significant word and doing binary search in that column until we get equality in that column. At that point, we move to the next column to the right and continue the binary search where we left off. Unfortunately, this does not quite work.

In Figure 10, which has $W/M = 3$, suppose we are searching for the three word identifier BMW (pretend each character is a word). We start by comparing in the leftmost column in the middle element (shown by the arrow labeled 1). Since the B in BMW matches the B at the arrow labeled 1 we move to the right (not shown) and compare the M in BMW with the N in the middle location of the second column. Since $N < M$, we do the second probe at the quarter position of the second column. This time the two M's match and we move rightward and we find W, but (oops!) we have found AMW, not BMW which we were looking for.

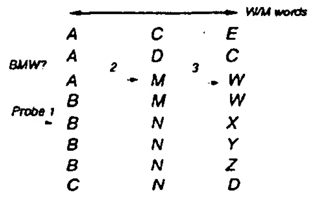


Fig. 10. Binary Search by columns does not work when searching for BMW

The problem is caused by the fact that when we moved to the quarter position in column 2, we assumed that all elements in the second quarter begin with B. This assumption is false in general. The trick is to add state to each element in each column which can contain the binary search to stay within a guard range.

In the figure, for each word like B in the leftmost (most significant) column, we add a pointer to the the range of all other words that also contain B in this position. Thus the first probe of the binary search for BMW starts with the B in BNX. On equality, we move to the second column as before. However, we also keep track of the guard range corresponding to the B's in the first column. The guard range (rows 4 through 6) is stored with the first B we compared.

.Thus when we move to column 2 and we find that M in BMW is less than the N in BNX, we attempt to half the range as before and try a second probe at the third entry (the M in AMT). However the third entry is lower than the high point of the current guard range (4 through 6). So without doing a compare, we try to halve the binary search range again. This time we try entry 4 which is in the guard range. We get equality and move to the right, and find BMW as desired.

In general, every multiword entry $W_1, W_2, \ldots, W_n$ will store a guard range with every word. The range for W_i, points to the range of entries that have $W_1, W_2, \ldots, W_i$ in the first i words. This ensures that when we get a match with W_i in the i-the column, the binary search in column $i+1$ will only search in this guard range. For example, the N entry in BNY (second column) has a guard range of $5-7$, because these entries all have BN in the first two words.

The naive way to implement guard ranges is to change the guard range when we move between columns. However, the guard ranges may not be powers of 2, which will result in expensive divide operations. A simpler way is to follow the usual binary search probing. If the table size is a power of 2, this can easily be implemented. If the probe is not within the guard range, we simply keep halving the range until the probe is within the guard. Only then do we do a compare.

The resulting search strategy takes $\log_2 N + W/M$ probes if there are N identifiers. The cost is the addition of two 16 bit pointers to each word. Since most word sizes are at least 32 bits, this results in adding 32 bits of pointer space for each word, which can at most double memory usage. Once again, the dominant idea is to use precomputation to trade a slower insertion time for a faster search.

We note that the whole scheme can be elegantly represented by a binary search tree with each node having the usual > and < pointers, but also an = pointer which corresponds to moving to the next column to the right as shown above. The subtree corresponding to the = pointer naturally represents the guard range.

B. Using Multicolumn and Multiway Search for IPv6

In this section we explore several possible ways of using the k-way search scheme for IPv6. With the 128 bit address, if we used columns of 16 bits each, then we would need 8 columns. With 16 bit keys we can do a 6-way search. So the number of memory accesses in the worst case would be log_6 (2N) + 8. For N around 50,000 this is 15 memory accesses. In general, if we used columns of M bits, the worst case time would be $log_{k+1} N + W/M$ where $W = 128$ for IPv6. The value of k depends on the cache linesize C. Since k keys requires $2k+1$ pointers, the following inequality must hold. If we use pointers that are p bits long,

$kM + (2k+1) * p \leq C$

For the Intel Pentium pro, C is 32 bytes, i.e. $32 * 8 = 256$ bits. If we use $p = 16$,

$k(M + 32) \leq 240$, with the worst case time being $log_{k+1} N + 128/M$.

In general, the worst case number of memory accesses needed is $T = \lceil (log_{k+1}(2N) \rceil + \lceil (W/m) \rceil$, with the inequality $Mk + (2k+1)p \leq C$, where N is the number of prefixes, W is the number of bits in the address, M is the number of bits per column in the multiple column binary search, k is the number of keys in one node, C is the cache linesize in bits, p is the number of bits to represent the pointers within the structure and T is the worst case number of memory accesses.

Fig 11 shows that the W bits in an IP address are divided into M bits per column. Each of these M bits make up a M bit key, k of which are to be fitted in the search node of length C bits along with $2k+1$ pointers of length p bits.

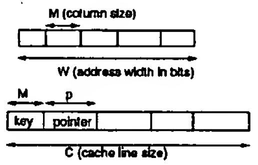


Fig. 11. Symbols used in expressing the number of memory accesses needed.

For typical values of N, the number of prefixes, the following table gives the value of the corresponding worst case number of memory accesses.

No. of Prefixes	$M = 16$	$M = 32$	$M = 64$
128	12	10	8
256	13	10	8
1024	14	11	9
50000	17	15	13
100000	17	16	14

TABLE IV

Worst case number of memory accesses for various values of N and M with $W = 128, C = 256$ (32 bytes) and $p = 24$ bits.

However, by using the initial array, the number of prefixes in a single tree can be reduced. For IPv4 the maximum number in a single tree was 336 for a practical database with N more than 30000 (i.e., the number of prefixes that have the same first 16 bits is 168, leading to 336 keys). For IPv6, with $p = 16$, even if there is an increase of 10 times in the number of prefixes that share the same first 16 bits, for 2048 prefixes in a tree we get a worst case of 9 cache line fills with a 32 byte cache line. For a 64 byte cache line machine, we get a worst case of 7 cache line fills. This would lead to worst case lookup times of less than

800 nsec, which is competitive with the scheme presented in [WVTP97].

C. Measurements

We generated random IPv6 prefixes and inserted into a k-way search with an initial 16 bit array. From the practical IPv4 database, it was seen that with N about 30000, the maximum number which shared the first 16 bits was about 300, which is about 1% of the total number of prefixes. To capture this, when generating IPv6 prefixes, we generated the last 112 bits randomly and distributed them among the slots in the first 16 bit table such that the maximum number that falls in any slot is around 1000. This is necessary because if the whole IPv6 prefix is generated randomly, even with N about 60000, only 1 prefix will be expected to fall in any first 16 bit slot. On a Pentium Pro which has a cache line of 32 bytes, the worst case search time was found to be 970 nsec, using M=64 and p=16.

VIII. Conclusion

We have described a basic binary search scheme for the best matching prefix problem. Basic binary search requires two new ideas: encoding a prefix as the start and end of a range, and precomputing the best matching prefix associated with a range. Then we have presented three crucial enhancements: use of an initial array as a front end, multiway search, and multicolumn search of identifiers with large lengths.

We have shown how using an initial precomputed 16 bit array can reduce the number of required memory accesses from 16 to 9 in a typical database; we expect similar improvements in other databases. We then presented the multiway search technique which exploits the fact that most processors prefetch an entire cache line when doing a memory access. A 6 way branching search leads to a worst case of 5 cache line fills in a Pentium Pro which has a 32 byte cache line. We presented measurements for IPv4. Using a typical database of over 30,000 prefixes we obtain a worst case time of 490 nsec and an average time of 130 nsec using storage of 0.7 Mbytes. We believe these are very competitive numbers especially considering the small storage needs.

For IPv6 and other long addresses, we introduced multicolumn search that avoided the multiplicative factor of W/M inherent in basic binary search by doing binary search in columns of M bits, and moving between columns using precomputed information. We have estimated that this scheme potentially has a worst case of 7 cache line fills for a database with over 50000 IPv6 prefixes database.

For future work, we are considering the problem of using different number of bits in each column of the multicolumn search . We are also considering the possibility of laying out the search structure to make use of the page mode load to the L2 cache by prefetching. We are also trying to retrofit our Pentium Pro with an SDRAM or RDRAM to improve cache loading performance; this should allow us to obtain better measured performance.

References

[Asc] Ascend. Ascend GRF IP Switch Frequently Asked Questions. http://www.ascend.com/299.html#15.

[BCDP97] Andrej Brodnik, Svante Carlsson, Mikael Degermark, and Stephen Pink. Small Forwarding Table for Fast Routing Lookups. *To appear in ACM Sigcomm'97*, September 1997.

[HP96] Hennessey and Patterson. *Computer Architecture A quantitative approach, 2nd Edn.* Morgan Kaufmann Publishers Inc, 1996.

[Inta] Intel. Intel Architecture Software developer's Manual, Vol 1: Basic Architecture. http://www.intel.com/design/litcentr/litweb/pro.htm.

[Intb] Intel. Pentium Pro. http://pentium.intel.com/.

[Mer] Merit. Routing table snapshot on 14 Jan 1997 at the Mae-East NAP. ftp://ftp.merit.edu/statistics/ipma.

[Mic] Micron. Micron Technology Inc. http://www.micron.com/.

[MTW95] Anthony J. Bloomfeld NJ McAuley, Paul F. Lake Hopatcong NJ Tsuchiya, and Daniel V. Rockaway Township Morris County NJ Wilson. Fast Multilevel heirarchical routing table using content-addressable memory. U.S. Patent serial number 034444. Assignee Bell Communications research Inc Livingston NJ, January 1995.

[NMH97] Peter Newman, Greg Minshall, and Larry Huston. IP Switching and Gigabit Routers. *IEEE Communications Magazine*, January 1997.

[oT] McGray Massachussetts Institute of Technology. Internet Growth Summary. http://www.mit.edu/people/mkgray/net/internet-growth-summary.html.

[Per92] Radia Perlman. *Interconnections, Bridges and Routers.* Addison-Wesley, 1992.

[PZ92] Tong-Bi Pei and Charles Zukowski. Putting Routing Tables in Silicon. *IEEE Network Magazine*, January 1992.

[Ram] Rambus. Rdram. http://www.rambus.com/.

[Sim] SimpleTech. Simple Technology Inc. http://www.simpletech.com/.

[Skl] Keith Sklower. A Tree-Based Routing Table for Berkeley Unix. Technical report, University of California, Berkeley.

[SV98] V. Srinivasan and George Varghese. Faster IP Lookups using Controlled Prefix Expansion. *Submitted to ACM Sigmetrics'98*, 1998.

[SW95] W. Richard Stevens and Gary R. Wright. *TCP/IP Illustrated, Volume 2 The Implementation.* Addison-Wesley, 1995.

[WVTP97] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable High Speed IP Routing Lookups. *To appear in ACM Sigcomm'97*, September 1997.

IEEE Xplore™ ◀ Back to Previous Page

Routing on longest-matching prefixes

Doeringer, W. Karjoth, G. Nassehi, M.
FH Worms, Germany;

This paper appears in: **Networking, IEEE/ACM Transactions on**

Publication Date: Feb. 1996
On page(s): 86 - 97
Volume: 4 , Issue: 1
ISSN: 1063-6692
Reference Cited: 28
CODEN: IEANEP
Inspec Accession Number: 5217324

Abstract:

This article describes the dynamic prefix tries, a novel data structure with algorithms for insertion, deletion, and retrieval to build and maintain a dynamic database of binary keys of arbitrary length. These tries extend the concepts of compact digital (Patricia) tries to support the storage of prefixes and to guarantee retrieval times at most linear in the length of the input key irrespective of the trie size, even when searching for longest-**matching prefixes**. The new design permits very efficient, simple and nonrecursive implementations of small code size and minimal storage requirements. Insert and delete operations have strictly local effects, and their particular sequence is irrelevant for the structure of the resulting trie, thus maintaining at all times the desired storage and computational efficiency. The algorithms have bees successfully employed in experimental communication systems and products for a variety of networking functions such as address resolution, maintenance and verification of access control lists, and high-performance **routing** tables in operating system kernels

Index Terms:

access protocols computer networks network operating systems operating system kernels telecommunication network **routing** tree data structures access control lists address resolution binary keys communication systems compact digital tries computational efficiency data structure deletion dynamic database dynamic prefix tries input key insertion longest-**matching prefixes** maintenance minimal storage networking functions operating system kernels retrieval **routing** small code size verification

Documents that cite this document
Select link to view other documents in the database that cite this one.

Routing on Longest-Matching Prefixes

Willibald Doeringer, *Member, IEEE*, Günter Karjoth, and Mehdi Nassehi, *Member, IEEE*

***Abstract*—This article describes the *dynamic prefix tries*[1]—a novel data structure with algorithms for insertion, deletion, and retrieval to build and maintain a dynamic database of binary keys of arbitrary length. These tries extend the concepts of compact digital (Patricia) tries to support the storage of prefixes and to guarantee retrieval times at most linear in the length of the input key irrespective of the trie size, even when searching for longest-matching prefixes. The new design permits very efficient, simple and nonrecursive implementations of small code size and minimal storage requirements. Insert and delete operations have strictly local effects, and their particular sequence is irrelevant for the structure of the resulting trie, thus maintaining at all times the desired storage and computational efficiency. The algorithms have been successfully employed in experimental communication systems and products for a variety of networking functions such as address resolution, maintenance and verification of access control lists, and high-performance routing tables in operating system kernels.**

I. Introduction

THE retrieval of context information identified by a sequence of binary digits, or binary keys, is an operation that is literally ubiquitous in computer networks [5], [24]. Applications range from the mapping of connection identifiers to state information for protocol processing of various kinds, to assigning and checking access or security tokens for users of shared network resources, to directory look-ups such as for address resolution purposes [7]. The particular example that initiated the design of our new algorithms stems from modern routing protocols. These protocols view addresses as unstructured binary sequences, and they distribute reachability information in the form of address prefixes, relaying data and, control packets according to the longest matching prefix rule [3], [4], [11], [12], [14], [19], [22]. That is, reachability information is distributed between routers in the form of variable-length binary sequences, each of which describes the potentially huge set of end-system addresses with the given binary sequence as prefix [5], [13], [24]. Given a destination address, a router bases its forwarding decisions on that address prefix among all those known to it, that constitutes the longest prefix of the destination address. Hence, a need arises to organize the address prefixes of which a router becomes aware over time into dynamic routing tables, or *forward information bases* [11], [26], such that searches for longest matching prefixes may be performed very efficiently.[2]

In many such applications, the respective databases of binary keys fluctuate in size rather dynamically, and the employed algorithms often execute in environments, such as operating system kernels in which:

- Code and storage efficiency is crucial.
- Strict stack limitations require nonrecursive and robust implementations.
- The processing time is at a premium, such that reliable upper bounds for the algorithmic complexity of all operations are required independent of the size of the database—most prominently so for the retrieval operation.

Our observations led us to derive the following list of requirements for a database of binary keys to be widely applicable in networking environments:

- Keys must be allowed to be of variable length and prefixes of each other.
- Operations must be provided for random insertion and deletion of keys without requiring recursion.
- The worst-case times for retrievals should have formally proven upper bounds independent of the size of the database, even for longest-matching prefix searches.
- Insertion and deletion operations should permit a flexible trade-off between performance and auxiliary information required in the database, and in all cases variants should be possible for which the execution time does not depend on the size of the database.
- The storage complexity should be kept to a minimum.
- Implementations that are simple and easy to understand should be possible.

A survey of applicable data structures and algorithms [10], [18] revealed that none of them meets all of our requirements without major modifications. Be it that no [9], [21] or only inadequate [2] delete operations are defined, support of prefixes is ruled out or inflicts significantly additional overhead [18], [21], nonrecursive implementations become intimidatingly complex [25], or the bounds for the algorithmic complexity depend on the size of the databases [27] or exhibit probabilistic nature [25], [27]. Hence, we decided to extend and improve the most efficient general binary trie structure known to date, the *Patricia tries* [1], [15], [16], [20], [21], [28], by defining a deletion operation that always restores the respective prior state of the trie and whose execution time is

Manuscript received June 30, 1995; approved by IEEE/ACM Transactions on Networking Editor J. Crowcraft.

W. Doeringer is with FH Worms, Germany (e-mail: w.doeringer@worms.fh-rpl.de).

G. Karjoth and M. Nassehi are with the IBM Research Division, Zurich Research Laboratory, Switzerland (e-mail: gka@zurich.ibm.com and mmn@zurich.ibm.com).

Publisher Item Identifier S 1063-6692(96)00610-3.

[1] *Trie* as in *retrieval*.

[2] Keys may comprise additional information for quality-of-service sensitive routing [26].

independent of the size of the database, and by integrating support for storing and retrieving (longest) matching prefixes without compromising the well-proven algorithmic efficiency [23] for the case when no prefixes are present.

The next section defines the data structures for *dynamic prefix tries (DP-Tries)*, the name of which was chosen to reflect the built-in support for random insertion and deletion of keys that may be prefixes of each other. Then we present the algorithms for insertion, deletion, and retrieval and prove their relevant properties. We conclude the article with a section on the performance of DP-Tries.

II. Data Structures And Definitions

The algorithms presented here operate on *(binary) keys*, defined as nonempty bit sequences of arbitrary but finite size. Keys are represented as $k = b_0, \cdots b_{l-1}$, with *elements* $k[i] = b_i \in \{0, 1\}$, $0 \le i \le |k|$, with $|k| = l - 1$ representing the *width* of k and $||k|| = l$ its *length*. Following standard terminology, a key k' is said to be a *(strict) prefix* of key k, denoted by $k' \preceq k$ ($k' \prec k$), iff $|k'| \le |k|$ ($|k'| < |k|$) holds, and $k'[i] = k[i]$, $0 \le i \le |k'|$. In particular, two keys are identical iff they are prefixes of each other. Keys are stored in *nodes*, each comprised of an index, at most two keys, and up to three links to other nodes. That is, a given node n has the following components [cf. Fig. 1(a)]:

Index(n)	The relevant bit position for trie construction and retrieval—the prefix to this bit position is common among all the keys in the subtrie that has node n as its root—the index allows the nonrelevant bits to be skipped, thus obviating one-way branches.
Leftkey(n)	A key with $LeftKey[Index(n)] = 0$ or NIL.
Rightkey(n)	A key with $RightKey[Index(n)] = 1$ or NIL.
Parent(n)	A link to the parent node of node n, NIL for the root node.
LeftSubTrie(n)	A link to the root node of that subtrie of node n with keys k such that $k[Index(n)] = 0$, or NIL.
RightSubTrie(n)	A link to the root node of that subtrie of node n with keys k such that $k[Index(n)] = 1$, or NIL.

Finally, a *dynamic prefix trie* is formally defined as any set of nodes and keys that form a binary trie with the properties listed in Lemma 3 of Section III.

To illustrate the data structures and the particular relation between nodes, we will now construct a DP-Trie for a sample set of keys $\{1000100, 1001, 10, 11111, 11\}$ using the graphical representation for nodes depicted in Fig. 1(a). When the first key, 1000100, is inserted into the trie, we obtain Fig. 1(b). The trie consists of only a root node, a. As key 1000100 is the only key of node a, the index takes its maximum value six, i.e., the width of the key. The key has a zero at bit position 6 and hence becomes the left key of the root. Fig. 1(c) depicts the DP-Trie after key 1001 is inserted. The index of node a

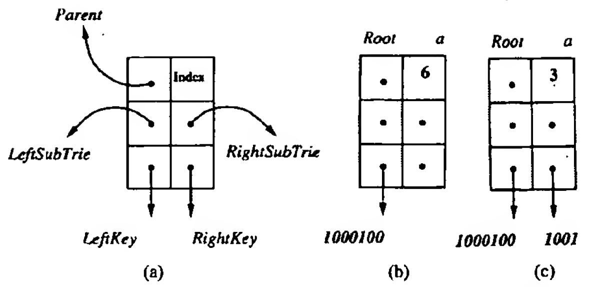


Fig. 1. Node structure and insertions: (a) node structure; (b) insertion of 1000100; (c) insertion of 1001.

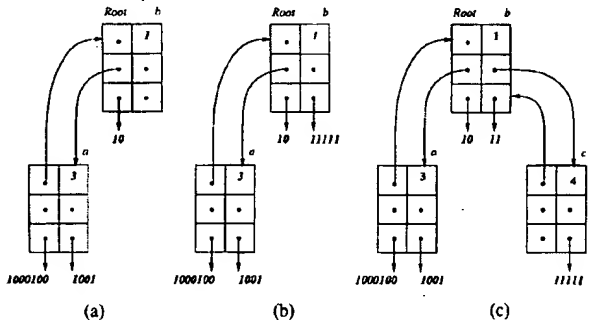


Fig. 2. More insertions: (a) insertion of key 10; (b) insertion of key 11111; (c) insertion of key 11.

becomes three, the first bit position at which the two keys differ, and the common prefix 100 is thus ignored. As the new key has a one in position 3, it becomes the right key of node a. (If the new key had a zero at this bit position and the old key a one, the old key would have been moved to become the right key and the new key would have become the left key.) In this trie, a search is guided to either stored key by the bit value at position 3 [*Index(a)*] of an input key; the bit positions 0, 1, and 2 are skipped. For example, a search with input 1001001111 returns 1001 as the longest matching prefix.

Adding key 10 results in Fig. 2(a). As key 10 is a prefix to both keys of node a, a new node, b, is created and made the parent of node a. The index of node b is one, the width of key 10. Key 10 and node a become, respectively, the left key and the left subtrie of node b, because all the stored keys have zero at position 1. Adding key 11111, we obtain Fig. 2(b). This key differs from key 10 in the bit position 1. So without a change in the index value, node b can accommodate the new key as its right key. Fig. 2(c) shows the DP-Trie after key 11 is added. Key 11 is a prefix of key 11111. Therefore, key 11111 is replaced by key 11 and pushed lower to the new node, c, that becomes the right subtrie of node b.

Now, let us consider some more search operations. A search for $k = 10011$ at node b is first guided to node a because $k[1] = 0$, and then to 1001; bit positions 0 and 2 are skipped. The latter key is identified as the longest matching prefix after a total of two bit and one key comparison, which are very basic and easily implemented operations in suitable hardware [23]. In contrast, a search for 100011 proceeds to node a only

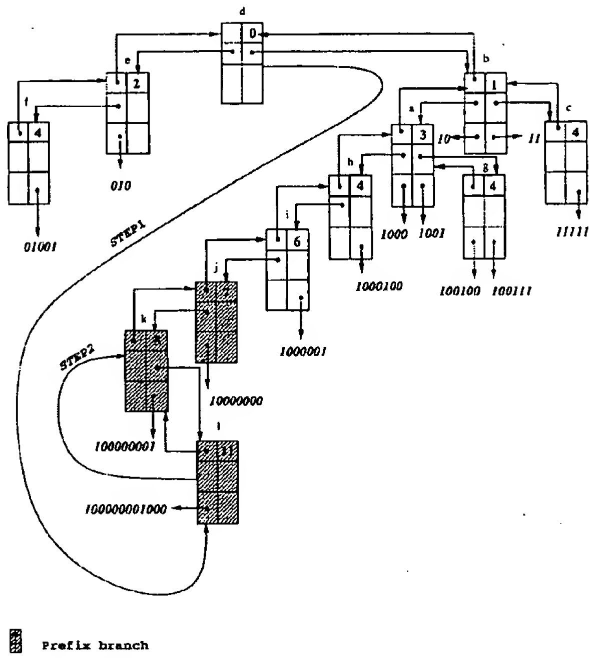


Fig. 3. Example of a DP-Trie.

to find that 1000100 is not a prefix of the input key. Using the *pointer reversal* provided by the parent links, the key 10 is identified as the longest matching prefix. Similarly, a search for key 001 branches to node a and terminates unsuccessfully at node b.

Further insertion of the keys depicted in Fig. 3 produces a sample DP-Trie that covers all structural aspects of general DP-Tries and will be used henceforth, as a reference. This figure can also be used to examine search operations involving more than a few nodes. For example, a search for key 1000011 is guided through nodes d, b, a, h, and i, skipping over bit positions 2 and 5. Then through pointer reversal, 1000 is identified as the maximum-length matching prefix.

III. Properties of DP-Tries

In this section, we describe the most relevant properties of DP-Tries that also establish the correctness of the algorithms defined in Section IV. The formal proof of the following results is given in Section VII, including the fact that the algorithms always preserve the stated properties. We begin our discussion with the central estimate for the storage complexity of DP-Tries.

Let us call a *Prefix Branch* any sequence of nodes $n_1, \cdots, n_k$, such that n_{i+1} is the root of the only subtrie of n_i for $1 \leq i \leq k-1$, and all nodes store exactly one key as strict prefixes of each other. Then we can derive the following result that shows the algorithms presented preserve the storage efficiency of Patricia tries in a naturally generalized way.

Lemma 1—Storage Complexity of DP-Tries: Let a^+ denote $Max(0, a)$ for a number a. The algorithms of a DP-Trie guarantee a minimal storage complexity described by the following relation, which is an optimal estimate for general DP-Tries: $\#nodes \leq \#keys + (\#PrefixBranches - 1)^+$.

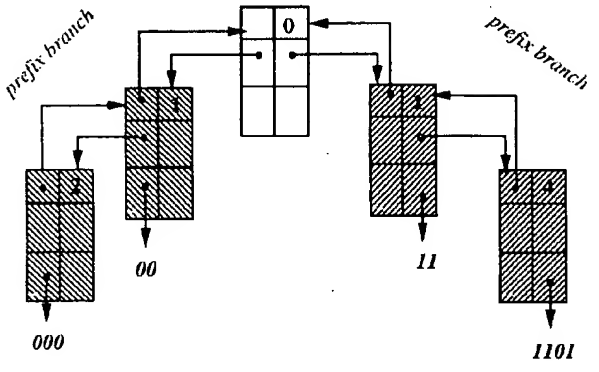


Fig. 4. Worst case storage complexity.

The example in Fig. 4 shows that the upper bound is optimal for DP-Tries, and the established results for Patricia tries follow from observing that in those tries, no prefix branches may exist.

The next result confirms that the structure of a DP-Trie is determined by the set of inserted keys only.

Lemma 2—Invariance Properties of DP-Tries:

1) For any given set of keys, the structure of the resulting DP-Trie is independent of the sequence of their insertion.
2) The delete operation always restores the respective prior structure of a DP-Trie.
3) The structure of a DP-Trie does not depend upon a particular sequence of insert or delete operations.

The correctness of the algorithms derives from the properties of DP-Tries stated in the next Lemma and the fact that they are invariant under the insert and delete operations (cf. Section VII).

Lemma 3—General Properties of DP-Tries:

N1 For a given node n, let $Keys(n)$ denote the set of keys stored in n and its subtries. All such keys have a width of at least $Index(n)$ and hence a length of at least $Index(n) + 1$.

N2 $Keys(n)$ denotes the subset of keys that share the respective common prefix of $Index(n)$ bits in length, denoted by $Prefix(n)$. That is, all keys in $Keys(n)$ match in bit positions up to and including $Index(n) - 1$. If $Index(n)$ is distinguishing, $Prefix(n)$ is of maximum length.

N3 The keys in any subtrie of node n share a common prefix of at least $Index(n)+1$ bits and are at least of such width.

N4 The index of a node n is maximal in the following sense: In a leaf node having only one key k, we have $Index(n) = |k|$, and in all other nodes there exist keys in $Keys(n)$ that differ at position $Index(n)$, or one of the keys stored in n has a width of $Index(n)$.

N5 Every node contains at least one key or two nonempty subtries. That is, a DP-Trie contains no chain nodes.

N6 If the left (right) key as well as the left (right) subtrie of node n are nonempty, then $|LeftKey(n)| = Index(n)$ ($|RightKey(n)| = Index(n)$).

N7 If the left (right) key of a node n is empty and the left (right) subtrie is not, then the subtrie contains at least two keys.

N8 For a given node n, the left (right) key and all keys in the left (right) subtrie of n have a 0 (1) bit at position $Index(n)$.

P1 $Prefix(n)$ increases strictly as n moves down a DP-Trie from the root to a leaf node, as does $Index(n)$. Hence, there are no loops and there always exists a single root node.

P2 Every full path terminates at a leaf node that contains at least one key, and there exists only one path from the root to each leaf node.

IV. Algorithms

This section provides a description of the algorithms for DP-Tries. They are represented in a notation that avoids distracting details, but is kept at a level that allows straightforward translations into real programming languages, such as C, as has been used by the authors for their implementations. The correctness of the algorithms follows directly from the properties of DP-Tries stated in Lemma 3 and the observation that these properties are invariant under the insertion and deletion operations as shown in Section III. We preface the description with the definition of a small set of commonly used operators, predicates, and functions. For convenience of representation, we shall generally denote a trie or subtrie by a pointer to its root node.

The first bit position where two keys k and k' differ is called their *distinguishing position*, and is defined as $DistPos(k,k') = Min\{i|k'[i] \neq k[i]\}$, with the convention that $DistPos(k',k) = |k'|+1$ for $k' \preceq k$. Furthermore, a node is said to be a *(single key) leaf node* if it has no subtries (and only stores one key). For example, in Fig. 2(c), node c is a single-key leaf-node, whereas node a is only a leaf-node, and $DistPos(101, 111000) = 1$.

The two operations that select the pertinent part of a node relative to an input key are defined below based on the convention that nonexisting subtries or keys are represented by the special value NIL:

```
Key(node, key) {
  if ($|key| < Index$(node)) then return(NIL)
  if ($key[Index$(node)]= 0) then return($LeftKey$(node))
  else return($RightKey$(node))
}

SubTrie(node, key) {
  if ($|key| < Index$(node)) then return(NIL)
  if ($key[Index$(node)] = 0) then return($LeftSubTrie$(node))
  else return($RightSubTrie$(node))
}
```

As an example, consider the nodes in Fig. 2(c). For $k = 0001$ we get as *SubTrie*(b, k) the trie starting at node a, and $Key(\mathrm{b,k}) = 10$.

A. Insertion

The insertion of a new key proceeds in three steps. First, the length of the longest common prefix of the new key with any of the keys stored in leaf nodes is identified (STEP1). Based on this information, the node around which the new key is to be inserted is located (STEP2). Finally, the key is added to the trie in the appropriate place (STEP3).

To accomplish the first step, we descend to a leaf node guided by the input key and make default choices where required. That is, as long as the width of the new key is greater than or equal to the respective nodal index, we follow the pertinent subtrie, if it exists. Otherwise we proceed along any subtrie, possibly with some implementation-specific (probabilistic) bias, until we reach a leaf node.[3] The keys in that node may then be used to calculate the longest common prefix (see Fig. 3 and Lemma 4). The node for insertion is identified by backtracking the downward path followed in the first step until a node is reached whose index is less than or equal to the length of the common prefix and the width of the input key, or the root node is reached (cf., Fig. 3). Depending on the prefix relations between the input key and those in the insertion node, the new key may then be inserted above [Fig. 2(a)], in [Fig. 1(b)], or below the selected node [Fig. 2(b)].

We now proceed to the description of the insertion algorithm, making use of the following auxiliary operations, that attempt to select subtries and keys of a node as closely related to an input key as possible. The procedures will be used to formalize the best effort descent of the first step.

```
ClosestKey(node, key) {
  if (|key| $\geq$ $Index$(node)) then
    /* A correct selection may be possible. */
    if ($Key$(node, key) $\neq$ NIL) then
      return($Key$(node, key))
    else
      /* Return the 'other' key. */
      return (Key(node, $BitComplement$(key)))
  else
    return (any key of node or NIL if none exists)
}

ClosestSubTrie(node, key) {
  if (|key| $\geq$ $Index$(node)) then
    /* A correct selection may be possible. */
    if ($SubTrie$(node, key) $\neq$ NIL) then
      return ($SubTrie$ (node, key))
    else
      /* Return the 'other' subtrie. */
      return ($SubTrie$(node, $BitComplement$(key)))
  else
```

[3]This step may be optimized. See Section IV-A4).

```
    return (any subtrie of node or NIL if none exists)
}
```

The operation that allocates and initializes new nodes of a DP-Trie is formalized as

```
Allocate Node(index, key) {
  local node
  NEWNODE(node) /* Allocate space for a new node */
  LeftKey(node) := RightKey(node) := NIL
  Parent(node) := LeftSubTrie(node) :=
    RightSubTrie(node) := NIL
  Index(node) := index; Key(node, key) := key
  return(node)
}
```

Given the above definitions, the algorithm for key insertion is defined in the following procedure. Further detailed comments on the processing and the storage requirements are summarized in Table I.

```
Insert(key) {
  local node, distpos, index

  /* Empty trie: Just add a node. Root is the global */
  /* pointer to the root node. */
  if ( Root = NIL ) then Root := AllocateNode(|key|, key)

  /* NonEmpty trie: Insertion proceeds in three steps. */
  else
    /* STEP1: Descend to a leaf node, identify the */
    /* longest common prefix, and the index of the */
    /* node to store the new key. Cp. Lemma 4. */
    node := Root /* Start at the Root */
    while (Not LeafNode(node))
      do node := ClosestSubTrie(node, key)
    distpos = DistPos(key, ClosestKey(node,key))
      /* Cp. Lemma 4. */
    index = $Min$(|key|, distpos)
      /* of the node to store the new key */

    /* STEP2: Ascend toward the root node, */
    /* identify the insertion node. */
    /* Cp. Lemma 4. */
    while (($index$ < $Index$(node))and(node $\neq$ Root))
      do node := Parent(node)

    /* STEP3: Branch to the appropriate insert operation. */
    /* See the following subsections */
    if (node = Root)
      InsertInOrAbove(node, key, distpos)
    elseif (SubTrie(node, key) = NIL) then
      InsertWithEmptySubTrie(node, key, distpos)
    else
      InsertWithNonEmptySubTrie(node, key, distpos)
}
```

The relevant properties of the local variable *distpos* and of the insertion node identified in STEP2 are summarized in the following Lemma.

Lemma 4—Properties of the Insertion Node: Let n denote the insertion node of STEP2 of the insertion algorithm, then:

a) The new key does not belong to any subtrie of n.
b) If $Min(\text{distpos}, |\text{key}|) \geq Index(n)$, then the key belongs to the trie starting at n.
c) If $Min(\text{distpos}, |\text{key}|) < Index(n)$, then n is the root and the new key needs to be inserted in or above n.
d) Distpos is the length of the longest common prefix of the insert key with any key stored in a leaf node of the trie.

1) InsertInOrAbove: This procedure deals with insertions above the root node. If the new key is not a prefix of a single key in the root node, it is simply added and the node index adjusted accordingly. Otherwise, the key is stored in a new node that becomes the root, and the current trie is added as its only subtrie.

```
InsertInOrAbove(node, key, distpos) {
  local newnode, index := $Min$(|key|, distpos)

  /* InorAbove-1: Add the new key to the root node. */
  if ((|key| $\geq$ distpos)and SingleKeyLeafNode(node)) then
    Index(node) := index
    Key(node, BitComplement(key)):=
      ClosestKey(node, key)
    Key(node, key) := key

  /* InorAbove-2: Add the new key in a new node above the root and */
  /* the current trie as its subtrie at the appropriate side of the new node */
  else
  newnode := AllocateNode(index, key)
  Parent(node) := newnode
  if (|key| $\geq$ distpos) then /* InorAbove-2.1: No prefix */
    SubTrie(newnode, BitComplement(key)) := node
  else /* InorAbove-2.2: The new key is a prefix of the stored keys! */
    SubTrie(newnode, key) :=node
  Root := newnode /* A new Root has been created. */ }
```

2) InsertWithNonEmptySubTrie: This procedure covers the cases of nonempty subtries at the insertion node. The new key is added if it fits exactly (see Lemma 3 [N6]), otherwise a new node with this key is inserted below the located node and above its subtrie. To improve the readability of the main algorithm, we introduce an additional subfunction that links two nodes as required for a given input key.

```
LinkNodes(node, subnode, key) {
  SubTrie(node, key) := subnode
  Parent(subnode) := node
}
```

```
InsertWithNonEmptySubTrie (node, key, distpos) {
  local newnode, subnode, index := $Min$(|key|, distpos)
  /* NonEmpty-1: The new key fits exactly. */
  if (|key| = Index(node)) then Key(node, key) := key
  /* NonEmpty-2: A new node is inserted below node */
  /* and above the subtrie */
  else
     subnode := SubTrie(node, key) /* Save pointer. */
     newnode := AllocateNode(index, key)
     Parent(SubTrie(node, key)) := newnode
     LinkNodes(node,newnode,key)

     /* Add the old subtrie at the appropriate side */
     if (|key| ≥ distpos) then
        /* NonEmpty-2.1: No prefix */
        if (SingleKeyLeafNode(subnode)) then
           /* Garbage collect this node! */
           Key(newnode,
              BitComplement(Key)) :=
              ClosestKey(subnode, key)
           DeallocateNode(subnode)
        else
           SubTrie(newnode,
                 BitComplement(Key)) :=
                 subnode
     else/* NonEmpty-2.2: New key is prefix */
        SubTrie(newnode, key) := subnode
}
```

3) InsertWithEmptySubTrie: This procedure performs the additions of keys in or below nodes whose pertinent subtrie is empty. It is the most complex of all insertion functions in that it must distinguish between five cases. The insertion node may not have a respective key stored, the new key and the stored key may be equal, the new or the stored key fits exactly and one of them needs to be stored in a new node below the current one, or, the most involved case, one key is a strict prefix of the other and both keys need to be placed into two separate nodes.

```
InsertWithEmptySubTrie (node, key, distpos) {
  local storedkey, dpos, newnode, newnewnode,
        index := Min(|key|, distpos)

  /* Empty-1: The new key may just be added to an existing node */
  if (Key(node, key) = NIL) then Key(node, key) := key

  /* Empty-2: The new key is a duplicate. */
  elseif (Key(node, key) = key) then
        Key(node, key) := key

  /* Empty-3: The stored key fits, and the new one */
  /* needs to be added below node. */
  elseif (|Key(node, key)| = Index(node)) then
     newnode := AllocateNode(|key|, key)
     LinkNodes(node,newnode,key)

  /* Empty-4: The new key fits, and the stored key */
  /* needs to be pushed down. */
  elseif (|key| = Index(node)) then
     newnode := AllocateNode (|Key(node, key)|,
                              Key(node, key))
     Key(node, key) := key
     LinkNodes(node,newnode,key)

  /* Empty-5: The stored key and the new key */
  /* are longer than the node index and need to */
  /* be stored below node */
  else
     storedkey := Key(node, key) /* Save the stored key. */
     Key(node, key) := NIL /* Will be moved. */
     dpos = DistPos(key,storedkey) /* Disinguishing position */

      /* Empty-5.1: The keys are not prefixes of each */
      /* other and may hence be stored in one node. */
      if (dpos ≤ Min(|key|, |storedkey|)) then
        newnode := AllocateNode(dpos, key)
        Key(newnode, storedkey) := storedkey
        LinkNodes(node,newnode,storedkey)

      /* Empty-5.2: The stored key is a strict prefix */
      /* of the new key: Each key is stored in a separate new node. */
      elseif (dpos > |storedkey|)) then
        newnode := AllocateNode(|storedkey|, storedkey)
        LinkNodes(node,newnode,storedkey)
        newnewnode := AllocateNode(|key|, key)
        LinkNodes(newnode,newnewnode,key)

      /* Empty-5.3: The new key is a strict prefix of */
      /* the stored key: Each key is stored in a separate new node. */
        newnode := AllocateNode(|key|, key)
        LinkNodes(node,newnode,key)
        newnewnode := AllocateNode (|storedkey|,
                                    storedkey)
        LinkNodes(newnode,newnewnode,storedkey)
}
```

4) Algorithmic Complexity: STEP3 of the algorithm has a complexity of $\mathcal{O}(1)$, and the first two steps are linear in the depth of the trie. In the absence of prefixes, the depth depends logarithmically on the number of keys (cf., Lemma 1 and [17]), whereas the general case may degenerate to a linear list of prefixes. However, by storing at each node the respective common prefix of all keys stored in that node and its subtries (cf., Lemma 3 [N2] in Section III) the first two steps may be combined and terminated, at the latest, when the insert key is exhausted. That is, the insert operation can be performed independent of the size of the database as a trade-off against a small increase in storage complexity. In all cases, STEP3 affects, at most, the insertion node, its parent or the root node of one of its subtries.

B. Deletion

To delete a key, we simply erase it from its node. However, the storage and computational efficiency of DP-Tries hinges on the fact that the structure of a DP-trie is determined solely by its keys and not by a respective sequence of insert and delete operations. Hence, the deletion algorithm needs to run a garbage-collection function in nodes from which keys have been deleted in order to restore the respective prior trie structure: If a node becomes empty after a key deletion, it is simply removed. Nodes that no longer store keys and have only one subtrie, so-called *chain nodes*, are also removed by linking their parent node and subtrie directly. When a node becomes a single-key leaf-node, its index is maximized by appropriately swapping its key. As a last step of the garbage-collection function, an attempt is made to move the key from a new single-key leaf node into its parent node. Further comments may be found in Table I.

```
DeleteKey(key) {
  local node, collnode := NIL, storedkey := NIL

   /* Step1: Search for the key to be deleted */
   node := Root
   if ((node $\neq$ NIL) and (|key| $\geq$ $Index$(node))) then
     while (($SubTrie$(node, key) $\neq$ NIL) and
           (|key| $\geq$ $Index$($SubTrie$(node, key))))
     do node := $SubTrie$(node, key)

  /* If node or the key could not be found return error */
  if ((node = NIL) or ($Key$(node, key) $\neq$ key))
           return(NotFound)

   /* Step2: Delete the key and garbage collect nodes */
   else
     $Key$(node, key) := NIL

     /* DelEmpty: Delete an empty node */
     if ($Empty$(node)) then
       /* DelEmpty-1: The Root is not deleted. */
       if (node $\neq$ Root) then
         $SubTrie$($Parent$(node), key) := NIL
         collnode := $Parent$(node)
       /* DelEmpty-2: The Root is deleted. The trie is now empty. */
       else Root := NIL
       $DeallocateNode$(node)

     /* DelChain: Delete chain nodes */
     elseif ($ChainNode$(node)) then
       if (node $\neq$ Root)
         $SubTrie$($Parent$(node), key) := $SubTrie$(node, key)
         $Parent$($SubTrie$(node, key)) := $Parent$(node)
         collnode := $SubTrie$(node, key)
       else
         Root := $SubTrie$(node, key)
         $Parent$(Root) := NIL
       $DeallocateNode$(node)

     /* DelMax: Maximize the index of single-key leaf-nodes */
     elseif ($SingleKeyLeafNode$(node)) then
       storedkey := $Key$(node, $BitComplement$(key))
       $LeftKey$(node) := $RightKey$(node) := NIL
       $Index$(node) := |storedkey|
       $Key$(node, storedkey) := storedkey
       collnode := node

     /* DelSingle: Handle a single-key subtrie */
     elseif ( ($SubTrie$(node, key) $\neq$ NIL) and
           $SingleKeyLeafNode$( $SubTrie$(node, key))) then
       collnode := $SubTrie$(node, key)

   /* Step3: Last step of garbage collection */
   /* DelGC: Attempt to move keys from single-key */
   /* leaf-nodes to the parent node */
   if ((collnode $\neq$ NIL) and
           $SingleKeyLeafNode$(collnode)) then
     storedkey := $ClosestKey$(collnode, key)

     if (($Parent$(collnode) $\neq$ NIL)) and
         ($Key$($Parent$(collnode), storedkey) = NIL)) then
       $Key$($Parent$(collnode), storedkey) := storedkey
       $SubTrie$($Parent$(collnode), storedkey) := NIL
       $DeallocateNode$(collnode)
}
```

Algorithmic Complexity: The search down the trie is linear in the length of the input key, and the removal of the key and the garbage collection of nodes has a complexity of $\mathcal{O}(1)$. Hence, the complexity of the delete operation does not depend on the size of the trie. The deletion has only a local impact on the trie structure in that it affects, at most, the node that stores the pertinent key and the node below and above it.

C. Search

The search algorithm performs a descent of the trie under strict guidance by the input key, inspecting each of its bit positions, at most, once. Once this first step has terminated, the traversed path is backtracked in search of the longest prefix. The decision not to perform key comparisons on the downward path resulted from a bias toward successful searches. In a networking environment, such as when performing routing decisions, negative results typically resemble error situations that cause data packets to be discarded and error messages to be sent with low priority. Notification or recovery of such errors is not deemed to be overly time critical [5], [24].

```
SearchKey(key) {
  local node := Root

  /* Check for empty tries and short keys */
  if ((node = NIL) or (|key| < $Index$(node))) then
          return(NIL)

  /* Step1: Downward path */
  while (($SubTrie$(node, key) $\neq$ NIL) and
          (|key| $\geq$ $Index$($SubTrie$(node, key))))
```

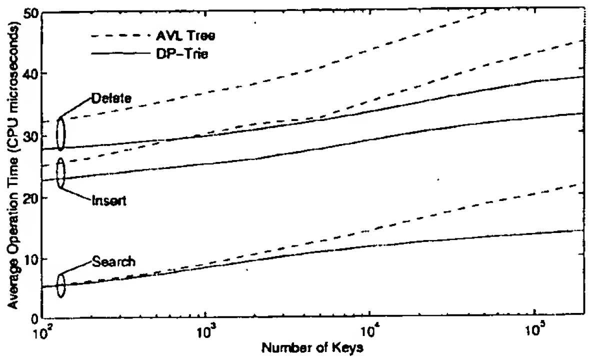


Fig. 5. Performance of DP-Trie and the AVL Tree for fixed-length keys.

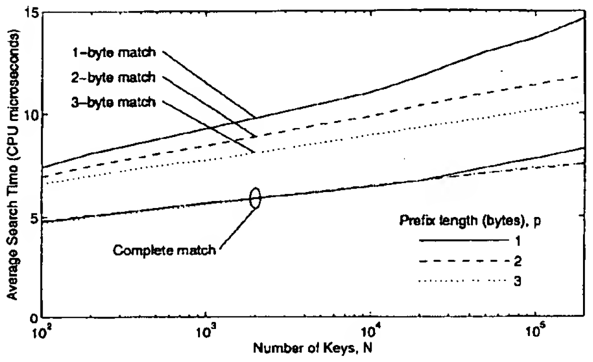


Fig. 6. Average search time in DP-Trie for different length-matching prefixes.

```
    do node := $SubTrie$(node, key)

    /* Step2: Backtracking to find the longest prefix */
    while (($node \neq$ NIL) and ($Key$(node, key) $\not\preceq$ key))
      do node := $Parent$(node)

    /* If a node was found, then it stores the longest prefix */
    if (node $\neq$ NIL) then return($Key$(node, key))
    else return(NIL)
}
```

Algorithmic Complexity: The complexity of the downward and upward searches are linear in the size of the input key independent of the size of the trie. In most implementations, the operations on the downward path will require only very little processing because only pointers need to be moved and single bits to be compared. On the upward path, the test for prefix relationships may be optimized for a given environment, such as by taking advantage of processor word sizes and instruction sets.

V. Performance

The performance of a DP-Trie is evaluated here in terms of the average time it takes to perform *insert*, *delete*, and *search* operations. The measurements are obtained using an implementation of the DP-Trie in ANSI C. In the case of fixed-length keys, a comparison is made with the performance of the well-known AVL tree [18]. The AVL tree algorithms used here were written by T. Bolmarcich of IBM Watson Research Center. Their performance closely matches that of another implementation reported in [25]. The performance results are in terms of the CPU time on an IBM RISC System/6000 model 970, which has a SPEC'92 integer performance of 47.8. The code was compiled using the *xlc* compiler at the optimization level 3.

Fig. 5 shows the mean *insert*, *delete*, and *search* times, in DP-Trie and AVL tree, as a function of the number of keys, where all keys are four bytes long. Here the keys were derived from uniformly distributed random integers (results were also obtained for keys based on a real set of IP addresses, but no significant difference was observed). Clearly, for all three operations, DP-Trie outperforms AVL tree. This indicates that the unique capability of DP-Trie to perform maximum-length prefix matching is not obtained at the cost of lower performance in the case of fixed-length keys. Also, detailed measurements indicate that the difference between the processing times of the three operations is almost entirely due to the time incurred in the *insert* and *delete* algorithms for, respectively, allocating and freeing space in memory. In the environment in which the measurements were performed, it was more costly to free space than to allocate it.

Let us consider the case where keys may be prefixes of each other. Fig. 6 shows the mean *search* time, in DP-Trie, as a function of the numbers of keys, where three key distributions are considered. Let N denote the number of keys in the Trie. The key distributions are parameterized by p, the size of prefix, for $p = 1, 2, 3$. More specifically, in the pth distribution, there are $N^{\frac{p}{4}}$ p-byte keys and N-$N^{\frac{p}{4}}$ four-byte keys. The p-byte keys are randomly generated and each one is used as the prefix of $N^{1-\frac{p}{4}}$–1 four-byte keys. The remaining 4-p bytes of four-byte keys are randomly generated.

Fig. 6 shows that the average search time increases linearly as a function of $\log(N)$. This indicates that the trees are quite balanced when there are prefixes. It can also be observed in the figure that the shorter the matching prefix, the longer it takes for the search operation to find it. This is a consequence of the fact that the *search* algorithm starts from the root, moves downward to a leaf and then backtracks to the first node with a matching prefix. Clearly, the shorter the matching prefix, the more nodes are traversed in the backtracking phase. In fact, because the tries are quite balanced, we can easily derive approximations that match the curves in Fig. 6 with high accuracy. Let c_0 denote the constant part of search time. In a balanced trie, the number of nodes traversed from the root to a leaf is $\log(N)$. Let c_1 denote the time spent at each node on the forward path. When there is no complete match, the number of nodes backtracked from the leaf to the node with the matching prefix is $\frac{(4-p)}{4}\log(N)$. Let c_2 denote the time spent backtracking each node, excluding the time spent for key comparisons. It turns out for the three key distributions considered here, when there is a complete match, exactly one key comparison is made. Otherwise, two key comparisons

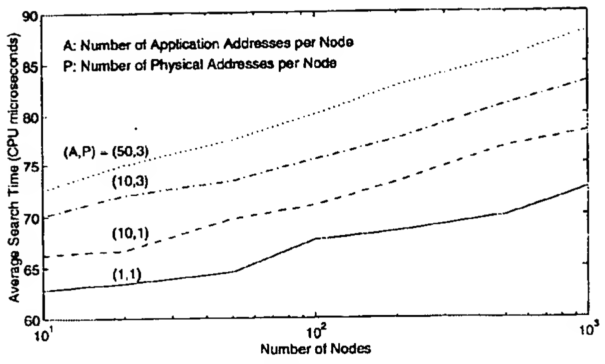


Fig. 7. Address database performance.

are made, first with a four-byte key and then with a p-byte key. Let c_3 denote the time it takes for a key comparison. Then the time of a search resulting in a p byte match is $c_0 + c_1 \log(N) + c_2(\frac{4-p}{4})\log(N) + 2c_3$ and the time of a search resulting in a complete match is $c_0 + c_1 \log(N) + c_3$. These approximations explain three features of Fig 6: a) the search time resulting in a complete match is independent of the length of prefix keys, p, so the bottom three curves are almost identical; b) the top three curves, that correspond to partial matches, are equally spaced, with a separation distance of $\frac{C_2}{4}\log(N)$; c) there is a wider gap between these curves and the bottom ones, namely $\frac{C_2}{4}\log(N) + c_3$.

As an application example, let us consider the performance of a dynamic address database for a heterogeneous internetworking environment built with DP-Tries as its base [6], [8]. This database maps between addresses of applications and those of the physical nodes they are installed on, all of which employ a common, general address format with three bit-strings of arbitrary and unrestricted sizes, i.e., a triple (*AddressFamily*, *HostAddress*, *LocalAddress*). Thus, each application address is mapped to one or more physical addresses owned by the same node, whereby no restrictions are placed on the mutual compatibility of the respective address components.

Fig. 7 shows the average search time in the address database as a function of the number of nodes in the network. It is assumed here that every node owns A applications and P physical addresses, where each of the node's application addresses are mapped to *all* of its P physical addresses. Each curve corresponds to a specific value of (A, P). As expected, the average search time increases logarithmically with the number of nodes, because the total number of keys, i.e., application addresses, is equal to A times the number of nodes. Let d_1 denote the rate of increase. The spacing between (1, 1) and (10, 1) curves is $d_1 \log(10/1)$ and that between (10, 3) and (50, 3) curves is $d_1 \log(50/10)$. The spacing between (10, 1) and (10, 3) is the constant representing the additional time for retrieving a list of physical addresses that contains three elements instead of one.

Comparing Figs. 5 and 7, we observe the average search time of the address database is about one order of magnitude larger than that of the basic DP-Trie. The reason behind this is the following. The address database evaluated here is part of an actual product. It was made highly reliable by consistency checking at various levels and provides additional functions such as group-address mapping, physical-address-to-application-address mapping and wildcard operations, that result in additional complexity.

The decision to implement the address database using DP-Trie instead of some other search data structures and algorithms was based on the following observations. DP-Trie is the only trie that handles variable-length keys and provides prefix matching. Although linked lists can provide these features, their performance is not acceptable, even for a relatively small number of keys. Furthermore, even for fixed-size keys, DP-Trie performs favorably compared to other tries, such as the AVL Tree.

VI. Conclusions

This article introduces a novel binary tree, referred to as DP-Trie, and associated algorithms that provide fast searches for longest matching prefixes with efficient storage utilization. No performance penalty is paid for fixed-length keys in that DP-Tries perform better than the state-of-the-art, as represented by AVL Tries.

DP-Tries have been successfully employed in experimental communication systems and products for a variety of networking functions, such as address resolution in a multiprotocol environment with a wide diversity of address formats, for the maintenance and verification of access control lists, and for high-performance routing tables in operating system kernels. With a minor modification to the search algorithm, DP-Tries can allow convenient access to *all* prefixes matching a given input key, and hence provide a natural implementation mechanism for the many network management applications operating on management information bases. Further applications of DP-Tries are being investigated, such as general alphabets with extensions for multiway branching, for efficient text retrieval, and for bibliographic searches.

VII. Proofs

This section presents the proofs of the results stated in Section III about DP-Tries. We start with Lemma 3, that lists the properties of DP-Tries establishing the correctness of the algorithms for insertion, deletion, and retrieval.

Proof of Lemma 3: The assertions in Lemma 3 follow from a straightforward induction argument on the number of inserted keys, and by verifying that the cited properties are preserved for each of the insertion cases (cf. Section IV-A) and that the deletion operation (cf. Section IV-B) restores the pertinent prior state of the trie. The full proof has been omitted here, and we refer the reader to Table I with comments on the algorithms of Section IV that contain all the arguments required to establish a rigorous proof. The table lists for each insertion case of Section IV-A the respective changes of the number of nodes and keys in a DP-Trie and summarizes the operations performed when a pertinent key is deleted. The

TABLE I
DETAILS

Insertion case	*Storage*		*Deletion case*
	Nodes	Keys	
EmptyTrie: Add the new key node with maximum index.	+1	+1	**DelEmpty-2**: Remove key, delete node. Flag trie as empty.
InorAbove1: Add the new key to the old node and adjust the index. The insertion node is the root node, which is a single-key leaf node here.	+0	+1	**DelMax**: Delete the key. Maximize the index in the single-key leaf node.
InorAbove2.1: The new key is not a prefix of the keys in the trie, and index denotes the first position where the keys differ.	+1	+1	**DelChain**: Delete the key. Remove the chain node (Root)
InorAbove2.2: In the prefix case, the trie is just linked to the new node.	+1	+1	**DelChain**: Delete the key. Remove the chain node (Root)
NonEmpty-1: Add the key.	+0	+1	**DelSingle**: Delete the key.
NonEmpty-2.1: As the new key is not a prefix of the subtrie keys, the key of an existing single key leaf node may be moved into the new node and the leaf node garbage collected.	+1	+1	**DelChain**: Delete the key. If the node is a chain node, remove it. If the result node is a single-key leaf node, maximize its index.
NonEmpty-2.2: Link subtrie to new node because keys are prefixes of each other.	+1	+1	**DelChain**: Delete the key. Remove the chain node.
Empty-1: Add the new key.	+0	+1	**DelMax**: Delete the key.
Empty-2: No change.	+0	+0	**DelMax**: Delete the key.
Empty-3: The new key is added to the new node with maximum index.	+1	+1	**DelEmpty-1**: Delete the key. Remove the empty node.
Empty-4: Move the old key into a new node below the insertion node.	+1	+1	**DelChain**: Delete the key. Move the key from the single-key subtrie leaf node to the current node and delete the subtrie.
Empty-5.1: The two keys are stored in a new node below the insertion node because they are not prefixes of each other.	+1	+1	**DelMax, DelGC**: Delete the key. Move the remaining key from the single-key leaf node to the parent node and delete the node.
Empty-5.2: The two keys are added in two separate nodes below the insertion node, thus forming a new prefix branch consisting of two nodes.	+2	+1	**DelEmpty-1, DelGC**: Delete key. Delete the empty node 'newnewnode'. Move the key from 'newnode' (single-key leaf node) into its parent node and delete 'newnode' also.
Empty-5.3: The two keys are added in two separate nodes below the insertion node, thus forming a new prefix branch consisting of two nodes.	+2	+1	**DelChain-1, DelGC**: Delete the key. Remove the chain node 'newnode'. Move the key from 'newnewnode' into its parent node and delete 'newnewnode' also.

latter information also shows that in each case the deletion operation restores the prior structure of the trie. □

The estimate of the storage complexity follows from Table I.

Proof of Lemma 1: The cases **Empty-5.2** and **Empty-5.3** are the only ones where two nodes have to be added to include one key and where prefix branches are formed. To see that the storage estimate of Lemma 1 also holds for these cases, we conclude first from Table I that always $\#Nodes \leq \#Keys + \#PrefixBranches$. However, the first time either case occurs, one may assume by Lemma 2 of Section III, and the fact that the width of the stored key is larger than the index of the insertion node, that the stored key was inserted last and into an already existing node. Hence we have prior to the insertion of the new key that $\#Nodes < \#Keys$, which implies that we need not count the very first prefix branch, and the result follows. The above argument no longer applies for any further occurrence of cases 5.2 and 5.3 because the insertion node may be the same node as in the previous case. □

We proceed with the proof of Lemma 4 about the properties of the insertion node of STEP2 of the insert operation.

Proof of Lemma 4: Let $nIns$ denote the insertion node and key the new key to be inserted. If the key belonged to a *subtrie* of $nIns$, we could deduce from N1 and N2 that $|key| \geq Index(\text{subtrie})$, and also $|distpos| \geq Index(\text{subtrie})$. In contradiction to the construction of STEP2 we would thus arrive at $Min(|key|, distpos) \geq Index(\text{subtrie})$. This proves a).

b) follows from N2 and the construction of STEP1.

To establish c), we first note that STEP2 implies for $Min(|key|, distpos) < Index(\text{nIns})$ that $nIns$ is the root node. For $|key| < Index(\text{nIns})$ assertion c) follows from N1. If $distpos < Index(\text{nIns})$ holds, then all keys in the trie match with the new key, at most, up to bit position $distpos-1$. Hence, the new key cannot be added to $Keys(nIns)$ by N1, and c) follows also for this case.

For a proof of d), it suffices to show that keys in nodes that are not traversed in STEP1 cannot have a longer common prefix with key. Assuming the contrary, we may infer from the monotonicity of $DistPos(\text{key},.)$that there exist two leaf nodes n, n' with keys k and k', such that $distpos = DistPos(k, key) < DistPos(k', key)$, with n denoting the leaf node from STEP1. Let n'' be the unique node where the paths from the root node to n and n' diverge (P2). Then we may conclude that $|key| \geq Index(n'')$ and by the construction of STEP1 that $k[Index(n'')] = key[Index(n'')] \neq k'[Index(n'')]$. However, this implies by N2 that $DistPos(k', key) = DistPos(k'', key) \leq DistPos(k, key)$ with $|k''| = Index(n'')$ and $k''[i] = k[i]$ for $0 \leq i \leq |k''|$, contradicting our assumption. □

We conclude this section with the proof of Lemma 2 about the invariance properties of DP-Tries.

Proof of Lemma 2: Assume that assertion (a) holds for sets of keys with cardinality less than or equal to $N \geq 1$. Let T and T' denote two DP-Tries built by two sequences of insertions of the same set of keys K of cardinality $N+1$. We want to show that $T \equiv T'$. We will do so by concluding that both tries have identical root nodes and then apply the induction hypothesis on their subtries.

By P1, we may identify root nodes R and R', respectively, and by N4 and P1, we see that $Index(R) = Index(R')$ holds. Let i denote this common index, i.e., $i := Index(R)$. If R contains no keys, N3 implies that all keys in K are of width $\geq i+1$. N5 and N7 imply that both subtries of R contain at least two keys each, and we may conclude from N6 that R' cannot contain any keys. However, the induction hypothesis and N8 imply that both subtries of R and R' are of the same structure, and hence, $T \equiv T'$ in this case.

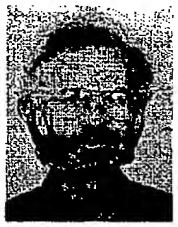
Assume now that *LeftKey*(R) is not empty. If *LeftSubTrie*(R) were empty, by N8 there exists only one key in K with a zero bit at position i. By N7, *LeftSubTrie*(R') is also empty and hence *LeftKey*(R')= *LeftKey*(R). In the case of a nonempty left subtrie, N3, N6, N8, and the induction hypothesis imply again that *LeftKey*(R')= *LeftKey*(R) and *LeftSubTrie*(R') $\equiv$ *LeftSubTrie*(R). Applying the above reasoning to the right key and subtries completes the proof of (a). (b) follows from the comments listed in Table I for the delete operation, and (c) is a direct consequence of (a) and (b). □

Acknowledgment

It is a pleasure to acknowledge the valuable comments provided by our colleague L. Svobodova on the initial draft of this paper. Thanks also go to the reviewers for their helpful comments, and to L.-M. Pavka for her editorial suggestions.

References

[1] A. Anderson, "On the balance property of Patricia tries: external path length viewpoint," *Theoretical Comput. Science*, vol. 106, no. 2, pp. 391–393, Dec. 1992.
[2] J. Aoe, K. Morimoto, and T. Sato, "An efficient implementation of trie structures," *Software Practice & Experience*, vol. 22, no. 9, pp. 695–721, Sept. 1992.
[3] ATM Forum, *PNNI Draft Specification*, 1995. Version 94-0471R10.
[4] S. Bradner and A. Mankin, *The recommendation for the IP next generation protocol*, RFC 1752, NIC, 1995.
[5] D. Comer, *Internetworking with TCP/IP. Principles, Protocols and Architecture*. Englewood Cliffs, NJ: Prentice-Hall, 1989.
[6] W. Doeringer *et al.*, "Multiprotocol transport networking: a software declaration of application independence," *IBM Systems J.*, vol. 34, no. 3, Sept. 1995.
[7] W. Doeringer, D. Dykeman, M. Peters, H. Sandick, and K. Vu, "Efficient, real-time address resolution in networks of arbitrary topology," in *Proc. 1st LAN Conf.*, 1993, pp. 183–191.
[8] W. Doeringer and M. Nassehi, "A new standard for address resolution," submitted to *IEEE Commun. Mag. Special Issue Enterprise Networking*, 1995.
[9] J. A. Dundas III, "Implementing dynamic minimal-prefix tries," *Software Practice & Experience*, vol. 21, no. 10, pp. 1027–1040, Oct. 1991.
[10] G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures*. Reading, MA: Addison-Wesley, 1991.
[11] ISO/IEC, *Intermediate System to Intermediate System Inter-Domain Routeing Exchange Protocol*, 1992, DIS 10747.
[12] ISO/OSI, *Network Service Definition, Addendum 2: Network Layer Addressing*, 1988, DIS 8348, Add.2.
[13] ——, *OSI Routeing Framework*, 1989, DIS 9575.
[14] ——, *Intermediate System to Intermediate System Intra-domain Routeing Protocol for Use in Conjunction with the Protocol for Providing Connectionless-mode Network Service (ISO 8473)*, 1990, DIS 10589.
[15] P. Kirschenhofer, H. Prodinger, and W. Szpankowski, "Do we really need to balance Patricia tries?" in *Proc. ICALP, Lecture Notes Comput. Science 317*, 1988, pp. 302–316.
[16] ——. "On the balance property of Patricia tries: external path length viewpoint," *Theoretical Computer Science*, vol. 68, no. 1, pp. 1–18, Oct. 1989.
[17] D. E. Knuth, "Optimum binary search trees," *Acta Informatica*, vol. 1, pp. 14–25, 1971.
[18] ——, *The Art of Computer Programming, vol. 3. Sorting and Searching*. Reading, MA: Addison-Wesley, 1991.
[19] K. Lougheed and Y. Rekhter, *Border gateway protocol (BGP)*, RFC 1163, NIC, 1990.
[20] T. X. Merrett and B. Fayerman, "Dynamic Patricia," in *Proc. Int. Conf. Foundations Data Org.*, Kyoto, Japan, 1985, pp. 13–20.
[21] D. R. Morrison, "Patricia—practical algorithm to retrieve information coded in alphanumeric," *J. ACM*, vol. 15, no. 4, pp. 515–534, Oct. 1968.
[22] P. Robinson, *Suggestion for a new class of IP addresses*, RFC 1375, NIC, 1992.
[23] T.-B. Pei and C. Zukowski, "Putting routing tables in silicon," *IEEE Network Mag.*, pp. 42–50, Jan. 1992.
[24] R. Perlman, *Interconnections: Bridges and Routers*. Reading, MA: Addison-Wesley, 1992.
[25] W. Pugh. "Skip lists: a probabilistic alternative to balanced tries," *Commun. ACM*, vol. 33, no. 6, pp. 668–676, June 1990.
[26] Y. Rekhter, "Forwarding database overhead for inter-domain routing," *ACM Comput. Commun. Rev.*, vol. 23, no. 1, pp. 66–81, Jan. 1993.
[27] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary tries," *J. ACM*, vol. 32, no. 2, pp. 652–686, July 1985.
[28] W. Szpankowski, "How much on the average is the Patricia trie better?" in *Proc. Allerton Conf.*, 1986, pp. 314–323.



Willibald Doeringer (M'95/ACM'95) received the Ph.D. degree in mathematics from the University Karlsruhe, Germany.

He is a faculty member of the computer science department of the Palatinate State College where he lectures in mathematics, computer science and communications. His recent research activities focus on internetworking and routing in large computer networks.

Günter Karjoth received the degree in computer science and the Ph.D. degree from the University of Stuttgart, Germany, in 1980 and 1987, respectively.

He was a Visiting Scientist at the Swedish Institute of Computer Science in Kista, Sweden, in 1986. Since 1986, he has been with the IBM Zurich Research Laboratory as a Research Staff Member. His research interest is in the modeling of distributed systems, their validation and implementation. He is currently working on access control issues in network management and object systems.

Dr. Karjoth has been a member of the ACM and of the German Computer Society (GI) since 1981.

Mehdi Nassehi (M'87) received the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 1987.

He has been with the IBM Zurich Research Laboratory since 1987. He is interested in network-based systems and currently works in the area of electronic commerce.

Dr. Nassehi served as an editor for IEEE TRANSACTIONS ON COMMUNICATIONS from 1990 to 1993.